Homework 10 Introduction to Big Data Systems

Christoffer Brevik

December 1, 2024

1 Code overview

To solve this weeks assignment I made 2 files with a C++ program utilizing GridGraph. My code is stored in the files `kcores.cpp` and `pagerank_delta.cpp`.

1.0.1 Running the Script

Running my Partitioning program can be done by going into the `./2024403421/hw10_src` folder in the computational node, or from the root folder of my delivery file. From here you will preprocess my data and run my mining algorithm. I've used both a Jupyter and a python file to process the seperate tasks of preprocessing and mining the graph. I will explain how to run both briefly here, but as these have the same contents I will focus on the *fsm.py* file moving forward:

1.0.2 Running the fsm.py

The *fsm.py* can be run by using the command:

```
python3 fsm.py
```

1.0.3 Running the fsm.ipynb

If you are using the code found in zipped folder using an IDE, or somehow can run Jupyter files on the computational node, you can also run the Jupyter file. How to run this depends on your IDE, but there should be a button to *Run all*.

2 My Code

All my code resides in the *fsm.py*, which is found in the root of the *hw10_src* folder. The file consists of two sections, preprocessing and mining, and I will explain both briefly below:

Preprocessing

The preprocessing section focuses on building a graph representation of the input data. This involves loading datasets, combining edges and vertices, and constructing an adjacency list representation for the graph.

- The data is read from multiple CSV files:
  - `account.csv` and `card.csv` are used to extract vertex data.
  - `account_to_account.csv` and `account_to_card.csv` are used to extract edge data.
- The vertex data is cleaned to ensure unique IDs by dropping duplicates, and it is set as the index for efficient lookups.
- The edge data is cleaned and standardized by rounding the transaction amounts (`amt`).
- Finally, the graph is represented as an adjacency list using a `defaultdict`, where each node points to a list of its neighbors and their associated attributes.

Here I chose to use a python `defaultdict` as using libraries like `networkx.DiGraph` proved to be way too slow. With the data cleaned and structured as one graph, we can start the mining process

Data Mining

The mining section identifies frequent subgraphs of a given size in the graph using a combination-based approach. The main steps include:

- Each edge is hashed to create a unique identifier based on its source, target, amount, strategy name, and buscode.
- The program iterates over all possible combinations of nodes to identify subgraphs of a specified size (`pattern_size`).
  - For each combination, edges between nodes are checked, and their attributes are collected if they exist.
  - A subgraph is considered valid if the number of collected edges matches the specified pattern size.
- Valid subgraphs are stored in a dictionary, where their hashed representation is used as the key. Subgraph frequencies are updated based on occurrences.
- Only subgraphs meeting the `support_threshold` are considered frequent. These subgraphs are saved in the resulting JSON file.

This approach ensures that the mining process is efficient and scalable for large datasets, leveraging hashing and dictionary-based frequency counting.

3 Results

Sadly my code did not finish in time, and I was unable to get the total number of subgraphs fulfilling the criteria. So I am unable to share my results as of this moment.

However, the program is designed to produce a JSON file containing frequent subgraphs that meet the threshold. Each entry in the JSON file includes:

- The frequency of the subgraph.
- The edges that make up the subgraph, including details like source, target, and attributes.